UNITED STATES PATENT APPLICATION FOR:

METHOD AND APPARATUS FOR LATENCY BASED THREAD SCHEDULING

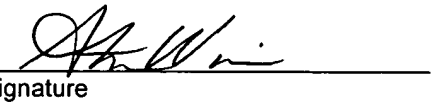
INVENTOR:

CURTIS R. PRIEM

ATTORNEY DOCKET NUMBER: NVDA/P000455

CERTIFICATION OF MAILING UNDER 37 C.F.R. 1.10

I hereby certify that this New Application and the documents referred to as enclosed therein are being deposited with the United States Postal Service on March 19, 2004, in an envelope marked as "Express Mail United States Postal Service", Mailing Label No. EL980273889US, addressed to: Commissioner for Patents, Box PATENT APPLICATION, Alexandria, VA 22313-1450.



Signature

Stephanie Winner

Name

March 19, 2004

Date of signature

CROSS-REFERENCE TO RELATED APPLICATION(S)

[0001] This application claims priority from commonly owned co-pending provisional United States Patent Application No. 60/458,191 entitled "METHOD AND APPARATUS FOR LATENCY BASED THREAD SCHEDULING," filed March 27, 2003, having common inventor and assignee as this application, which is incorporated by reference as though fully set forth herein.

FIELD OF THE INVENTION

[0002] The present invention relates to a novel method and apparatus for scheduling threads. More specifically, the present invention provides a method that schedules threads based on latency.

BACKGROUND OF THE DISCLOSURE

[0003] In programming, a "thread" can be thought of as a minimum "unit of execution" or a basic unit of CPU utilization. The thread model is a flexible process organization mechanism that allows individual processes to access common resources, thereby increasing program efficiency. Namely, a process that has multiple threads of control can perform more than one task at a time. For example, on a single processor machine, this means processing threads on a time-interleaved basis, thereby requiring scheduling. Although the thread model provides the benefit of concurrency of processes, it is up to the operating system or kernel to perform thread scheduling to properly exploit this concurrency.

[0004] For example, most operating systems schedule threads to handle interrupts by simply placing them into a queue. The threads are then processed in the order that the threads were scheduled (e.g., threads scheduled to be performed in the order the corresponding interrupts were received). However, this approach does not distinguish the priority of the underlining processes that are performed by the threads in the queue. For example, interrupt threads will be mixed with non-interrupt threads for scheduling.

[0005] To address the issue of priority, some advanced operating systems have two queues, where one queue is dedicated to high priority processes and one queue is dedicated to address low priority processes. In this approach, an interrupt thread can be scheduled by being placed in either queue. The threads in the queues are then processed in the order that they were placed in each queue. Although this approach allows the operating system some flexibility in prioritizing the execution of threads, it is still insufficient to allow the operating system to finely schedule threads to account for other parameters, e.g., latency requirement of a particular process.

[0006] For example, some real-time devices have "isochronous" requirements, where isochronous denotes "at the same rate", e.g., where a data flow is tied to time. To illustrate, from the perspective of a hardware device that is receiving a stream of real-time data (e.g., audio and video), the hardware device must handle a plurality of interrupts efficiently to address various functions such as buffer management (e.g., detecting fullness of a first buffer, setting up a second buffer, switching from the full buffer to an available empty buffer and the like). In other words, there is a fixed time limit when the first buffer is full and the second buffer must be ready to receive the continuous data stream. Since the data stream is directed toward the hardware device in real time, the hardware device must handle the interrupts skillfully to avoid a "pop" in the audio or a noticeable "glitch" in presenting the video if data packets are lost due to the inability of the hardware device to keep up with the continuous data stream.

[0007] Unlike a "closed" system where timing constraints are known a priori, the above timing criticality is further amplified in an open system, where additional hardware devices or boards can be optionally added. The operating system (OS) for the open system must skillfully balance the timing constraints of these different hardware devices so that they can all operate properly without accidental loss of data due to improper handling of interrupt events.

[0008] Additionally, operating systems that are designed to distinguish high priority events from low priority events are often defeated by selfish motives of hardware manufacturers. Specifically, device drivers for hardware devices that

are deployed in an open system are often written such that their interrupts will always be considered by the operating system to be high priority. Since each hardware device is competing for service by the operating system, there is little incentive to design a device driver that will cause its interrupts to be considered by the operating system as being low priority.

[0009] Additionally, even if interrupts from competing hardware devices are truly equal in terms of being in the same priority class, the operating system will not have any additional information to discriminate between numerous interrupt events. In such instances, the interrupts will likely be processed in the order that they are received in the queue, thereby limiting the ability of the operating system to flexibly rearrange the scheduled threads.

[0010] Therefore, a need exists for a novel method and apparatus that provides a thread scheduling approach that allows the operating system to finely schedule threads to account for thread parameters or attributes such as latency of a process and the like.

SUMMARY OF THE INVENTION

[0011] In one embodiment of the present invention, a novel method and apparatus for providing latency based thread scheduling is disclosed. Specifically, a thread attribute, e.g., latency of a process, is used in effecting the scheduling of the thread.

[0012] Using interrupt handling as an example, if an interrupt requests that a thread be scheduled to process the interrupt, then the interrupt scheduler will pass a latency value (e.g. in usec) to the operating system. The operating system will, in turn, apply this thread attribute to correctly organize all pending threads in an efficient order such that a thread that needs to be processed the soonest will be scheduled first.

[0013] In one embodiment, the operating system may take the latency value and add it to the current time to acquire an absolute time that the thread will need to be woken up. Using this novel approach, low priority processes, e.g., non-

interrupt based scheduled threads, will have a very high (e.g. infinite) latency value such that these low priority threads will always be scheduled after high priority threads (e.g., interrupt threads). This novel approach provides a very finely tuned thread scheduling method, where the operating system can exploit a thread attribute such as latency in arranging the order in which pending threads will be processed.

BRIEF DESCRIPTION OF THE DRAWINGS

[0014] The teachings of the present invention can be readily understood by considering the following detailed description in conjunction with the accompanying drawings, in which:

[0015] FIG. 1 illustrates a block diagram of a system for providing latency based thread scheduling of the present invention;

[0016] FIG. 2 illustrates an alternate block diagram of a system for providing latency based thread scheduling of the present invention;

[0017] FIG. 3 illustrates a time line diagram for scheduling a latency based thread; and

[0018] FIG. 4 illustrates a flowchart for a method for scheduling a latency based thread.

[0019] To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

DETAILED DESCRIPTION

[0020] FIG. 1 illustrates a block diagram of a system for providing latency based thread scheduling of the present invention. Specifically, FIG. 1 illustrates a system 100 of the present invention as implemented using a general purpose computer. The computer system 100 comprises a processor (CPU) 110, a

memory 140, e.g., random access memory (RAM), a read only memory (ROM) 130, and various input/output devices 120, (e.g., storage devices, including but not limited to, a tape drive, a floppy drive, a hard disk drive, a compact disk (CD) drive or a digital videodisk (DVD) drive, a receiver, a transmitter, a speaker, a display, a speech signal input device, e.g., a microphone, a keyboard, a keypad, a mouse, an A/D converter, a chipset, a controller card, a graphics card, a sound card, and the like).

[0021] In brief, an operating system 150, device drivers 160 and user applications 170 are loaded into the memory 140 and operated by the CPU 110. In one embodiment, the operating system 150 comprises a plurality of software modules, including by not limited to a thread scheduler 152 and an interrupt handler 154. The thread scheduler 152 is employed to schedule threads for supporting multithreaded applications, whereas the interrupt handler 154 is employed to implement an interrupt service routine. It should be noted that although the thread scheduler 152 and the interrupt handler 154 are illustrated as separate modules, the present invention is not so limited. Namely, those skilled in the art will realize that the functions performed by these modules can be implemented in a single module, e.g., within an "interrupt scheduler", or serving as a part of a much larger software module.

[0022] In operation, a thread attribute, e.g., latency of a process, is used in affecting the scheduling of the thread. Although the present invention is described below within the context of coordinating thread scheduling to handle interrupts, those skilled in the art will realize that the present invention can be adapted to address thread scheduling in general. Any multithreaded real-time computing environment can make use of this invention. The invention is particularly useful for multimedia applications.

[0023] Using interrupt handling as an example, if an I/O device 120 issues an interrupt request via a device driver 160, the thread scheduler 152 will schedule a thread to process the interrupt. Specifically, in one embodiment of the present invention, the interrupt handler 154 or optionally, the device driver 160, will pass a latency value (e.g. in usec) to the operating system 150. The operating

system will, in turn, apply this thread attribute to correctly organize all pending threads in an efficient order such that a thread that needs to be processed the soonest will be scheduled first.

[0024] In one embodiment, the operating system may take the latency value and add it to the current time to acquire an absolute time that the thread will need to be woken up and processed. Using this novel approach, low priority processes, e.g., non-interrupt based scheduled threads, will have a very high (e.g. infinite) latency value such that these low priority threads will always be scheduled after high priority threads (e.g., interrupt threads). This novel approach provides a very finely tuned thread scheduling method, where the operating system can exploit a thread attribute such as latency in arranging the order in which pending threads will be processed.

[0025] Thus, it should be understood that the thread scheduler 152, interrupt handler 154 and the device drivers 160 can be implemented as one or more physical devices (e.g., registers and ROMs) that are coupled to the CPU 110 through a communication channel. Alternatively, the thread scheduler 152, interrupt handler 154 and the device drivers 160 can be represented by one or more software applications (or even a combination of software and hardware, e.g., using application specific integrated circuits (ASIC)), where the software is loaded from a storage medium, (e.g., a magnetic or optical drive or diskette) and operated by the CPU in the memory 140 of the computer. Alternatively, the thread scheduler 152, interrupt handler 154 and the device drivers 160 can be represented by Field Programmable Gate Arrays (FPGA) having control bits. As such, the thread scheduler 152, interrupt handler 154 and the device drivers 160 (including associated methods and data structures) of the present invention can be stored on a computer readable medium, e.g., RAM memory, magnetic or optical drive or diskette and the like.

[0026] FIG. 2 illustrates an alternate block diagram of a system 200 for providing latency based thread scheduling of the present invention. In one exemplary embodiment, FIG. 2 illustrates the specific interaction between the operating

system or kernel 210 and various hardware devices 220a-b and software applications 270.

[0027] FIG. 2 also illustrates a user boundary (dashed line) where the user applications 270, application programming interface 272 and various software drivers 274 are typically accessible by the user of the overall system, whereas the region below the dashed line can be broadly perceived as being on the "kernel and hardware" side that is not accessible by the user. It should be noted that although the present latency based thread scheduling will be disclosed below in terms of hardware interrupts, the present invention is not so limited. Those skilled in the art will realize that the present invention is equally applicable to the scheduling of threads that are generated in response to a software application 270.

[0028] FIG. 2 illustrates an operating system kernel 210 that interfaces with a device driver or portion of a device driver 260. In one embodiment, the device driver 260 is implemented with an OS dependent code or module 262, an NvArch code or module 264 and a hardware dependent code 266a (shown embodied within a ROM). The hardware dependent code 266a tracks a corresponding hardware device, e.g., a chip1 220a. As such, a second hardware dependent code 266b within a ROM is deployed to track a second hardware device, e.g., a chip2 220b and so on if additional hardware is deployed.

[0029] Unique changes or operational constraints affecting a particular chip will be handled and noted by its corresponding hardware dependent code 266. Thus, the hardware dependent code 266 can accurately identify the latency associated with its own hardware device. This is due to the fact that latency may be associated with clock rate, display rate, buffer size and so on. These important information are tracked by the hardware dependent code 266 stored in a ROM in one embodiment. The use of this latency information to effect latency based thread scheduling will be further described below.

[0030] Each of the hardware devices is provided with a direct interrupt line 222 to the OS kernel 210 (or more accurately, to the controller circuitry that is in communication with the OS kernel). Once the interrupt line 222 is set by a hardware device, a series of communication will follow between the device driver 260 and the OS kernel 210. In one embodiment, this communication is implemented via various "calls".

[0031] The first call is known as "INTERRUPT_ALLOC" (nvOsCallback(NvOs8a0 Parameters *)) which is a callback to allocate an interrupt. The allocation nvOsCallback() normally occurs when a device is allocated for the first time, e.g., when the hardware is being initialized.

[0032] In operation, the NV Architecture code 264 calls the operating system dependent code 262 to allocate a real-time interrupt. Normally the operating system dependent code 262 allocates appropriate data structures and then connects the interrupt. The NV Architecture allows multiple interrupts per device since each device contains many real-time engines. Some non-real-time operating systems can only schedule one thread or one thread per device. Real-time operating systems should create a thread for each interrupt allocated. The operating system 210 should attempt to service each allocated interrupt separately to ensure more predictable results.

[0033] The NV Architecture can make an nvOsCallback() with the following parameter members:

NvOs8a0Parameters

This data structure contains the input and output parameters for the nvOsCallback() function.

"function"

This field contains the function number.

The argument is a 32 bit void number. Legal argument values are:

0x000008A0 NVOS 8A0_INTERRUPT_ALLOC

The argument is returned unchanged.

"status"

This field returns the status.

The argument is a 32 bit void number. All argument values are legal.

Possible returned values are:

0x00000000 NVOS 8A0_STATUS_SUCCESS

0x00000001 NVOS 800_STATUS_ERROR_BAD_FUNCTION

0x00000002

NVOS8A0_STATUS_ERROR_INSUFFICIENTRESOURCES

0x00000003 NVOS8A0_STATUS_ERROR_BAD_INTERRUPT_ID

"interruptId"

This field contains the identification number of the interrupt.

The argument is a 32 bit void number. The argument values must not be equal to the interruptId of a previously allocated interrupt.

The argument is returned unchanged.

[0034] The second call is known as "INTERRUPT_FREE" (nvOsCallback(NvOs8a1 Parameters *)) which is a callback to free an interrupt. Specifically, the NV architecture code 264 calls the operating system dependent code to free a real-time interrupt. The free nvOsCallback normally occurs when the last usage of a device is freed. Namely, if a hardware device is being shut down, it is necessary to free up all those resources and data structures that were created and validated for the hardware device.

NvOs8a1Parameters

This data structure contains the input and output parameters for the nvOsCallback() function.

"function"

This field contains the function number.

The argument is a 32 bit void number. Legal argument values are:

0x000008A1 NVOS 8A1_INTERRUPT_FREE

The argument is returned unchanged.

"status"

This field returns the status.

The argument is a 32 bit void number. All argument values are legal.

Possible returned values are:

0x00000000 NVOS 8A1_STATUS_SUCCESS

0x00000001 NVOS 800_STATUS_ERROR_BAD_FUNCTION

0x00000002 NVOS8A1_ STATUS_ERROR_BAD_INTERRUPT_ID

"interruptId"

This field contains the identification number of the interrupt.

The argument is a 32 bit void number. The argument values must be equal to the interruptId of a previously allocated interrupt.

The argument is returned unchanged.

[0035] The third call is known as "INTERRUPT_SCHEDULE" (nvOsCall(NvOs0a0 Parameters *)) which is a call to schedule interrupts. This call is an nvOsCall() from the operating system dependent code 262 to the NV Architecture code 264 to schedule interrupts. Specifically, the nvOsCall() should be made when the operating system 210 notifies the operating system dependent code 262 that a device 220 may have some pending interrupts that require servicing. In advanced operating systems, a thread should be scheduled with a priority corresponding to NvOs0a0 Parameters.latency for each interrupt that needs servicing. INTERRUPT_SCHEDULE should be continuously called until NvOs0a0 Parameters.status returns ERROR_NO_MORE_INTERRUPTS.

NvOs0a0Parameters

This data structure contains the input and output parameters for the nvOsCall() function.

"function"

This field contains the function number.

The argument is a 32 bit void number. Legal argument values are:

0x000000A0 NVOS 0A0_INTERRUPT_SCHEDULE

The argument is returned unchanged.

"status"

This field returns the status.

The argument is a 32 bit void number. All argument values are legal.

Possible returned values are:

0x00000000 NVOS0A0_STATUS_SUCCESS

0x00000001 NVOS000_STATUS_ERROR_BAD_FUNCTION

0x00000002 NVOS0A0_

STATUS_ERROR_NO_MORE_INTERRUPTS

"interruptId"

This field contains the identification number of the interrupt.

The argument is a 32 bit void number. All argument values are legal.

The argument returns the identification number of the interrupt to be scheduled.

"latency"

This field returns the maximum latency until the service nvOsCall() must be called or else the hardware device's real-time engines may not be able to keep up with isochronous data streams.

The argument is a 32 bits unsigned integer, measured in nanoseconds. All argument values are legal.

The argument returns maximum latency in nanoseconds of the service routine.

[0036] The fourth call is known as "INTERRUPT_SERVICE" (nvOsCall(NvOs0a1 Parameters *)) which is a call to service interrupts. This call is an nvOsCall() from the operating system dependent code 262 to the NV Architecture code 264 to service an interrupt. Specifically, the nvOsCall() should be used to request servicing of the specified interrupt as determined by a previous call to INTERRUPT_SCHEDULE. Namely, the operating system is simply calling to each device driver 260 to go ahead and process your interrupts.

NvOs0a1Parameters

This data structure contains the input and output parameters for the nvOsCall() function.

"function"

This field contains the function number.

The argument is a 32 bit void number. Legal argument values are:

0x000000A1 NVOS 0A1_INTERRUPT_SERVICE

The argument is returned unchanged.

"status"

This field returns the status for this operation.

The argument is a 32 bit void number. All argument values are legal.

Possible returned values are:

0x00000000 NVOS0A1_STATUS_SUCCESS

0x00000001 NVOS000_STATUS_ERROR_BAD_FUNCTION

0x00000002 NVOS0A1_STATUS_ERROR_BAD_INTERRUPT_ID

"interruptId"

This field contains the identification number of the interrupt.

The argument is a 32 bit void number. The argument values must be equal to the interrupted of a previously allocated interrupt.

The argument is returned unchanged.

[0037] It should be noted that the various calls and their structures as disclosed above are only illustrative. In fact, the specific structure of the device driver 260 as having an OS dependent code 262, an NV Architecture code 264 and a hardware dependent code 266 are illustrative of one particular embodiment. Those skilled in the art will realize that these modules or codes can be implemented in different manners, e.g., being partially or wholly incorporated within the OS kernel 210.

[0038] FIG. 3 illustrates a time line diagram for scheduling a latency based thread. In describing this time line, FIG. 3 can also serve as a flowchart for a method for scheduling a latency based thread.

[0039] Specifically, FIG. 3 illustrates a time line diagram that describes the scheduling of a latency based thread for swapping buffers, e.g., as in a vertical retrace operation. FIG. 3 illustrates at time line 310 where buffer A is being read and is reaching the point of being emptied at point 312, where the read operation transitions from buffer A to buffer B as shown in time line 320. Namely, at point 312, buffer A is completely emptied and data is now being read from buffer B instead.

[0040] In time line 330, the hardware device triggers an interrupt to the OS, e.g., via interrupt line 222a. Responsive to the hardware device, the OS in time line 340 toggles all the other interrupt lines 222b-n to the "off" position briefly to record the interrupt number on interrupt line 222a. During this short period, the OS will not acknowledge any other interrupt lines. Once the interrupt information is captured, the OS will toggle all the interrupt lines back to the "on" position so that it can detect and receive the next interrupt. This approach accords the OS the ability to minimize the amount of time where interrupt lines are actually turned off.

[0041] It should be noted that most CPUs have two interrupt request lines. One is the "nonmaskable" interrupt, which is reserved for events such as unrecoverable memory errors. The second interrupt line is "maskable", where it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is typically used by device controllers to request service. Thus, the present invention only masks interrupts for the short period during of logging of the interrupt events, minimizing the likelihood of that a maskable or nonmaskable interrupt will be received and ignored. Furthermore, subsequent processing steps such as computing a latency value or the scheduling of a thread are performed without masking interrupts. In some embodiments of the invention, nonmaskable interrupts are not masked during the logging of the interrupt events.

[0042] In time line 350, the hardware dependent code will acquire and provide latency information or a latency value associated with setting up the next buffer. This latency information can be a predefined value associated with a particular

interrupt number or it can be calculated in accordance with an equation. In the present example, the equation can be formulated as having parameters associated with "buffer size", "sampling rate", "data transmission rate" and the like.

[0043] In time line 360, the latency information provided by the hardware dependent code is used by the thread scheduler of the operating system to schedule the thread to be serviced at some future time. In other words, the thread scheduler 152 has now exploited the latency information such that interrupts are not necessarily serviced in the order that they are received by the operating system. In fact, the thread scheduler has the necessary information to now rearrange various threads in the queue such that threads that require faster service based on latency will be processed ahead of other threads that can wait a bit longer.

[0044] In time line 370, the thread is actually awoken to perform the interrupt service. Finally, time line 380 illustrates various time segments that can be used to formulate the latency information. For example, the time duration between points "a-g" can be used as the latency information in scheduling the thread because it represents the maximum time allowed before buffer B will be emptied and the read operation will switch back to buffer A. Alternatively, the time duration between points "e-f" can be used as the latency information in scheduling the thread because it represents the time necessary to service the interrupt, i.e. process the thread. Yet another alternative is to use the time duration between points "a-d" because it represents the time duration that is necessary to setup a thread to service the interrupt. In fact, the latency information can be derived from various segments of this time line or other criteria to implement the present latency based thread scheduling.

[0045] FIG. 4 illustrates a flowchart for a method for scheduling a latency based thread. In step 405 an I/O device, such as I/O device 120 issues an interrupt request via a device driver, such as device driver 16. The interrupt request is received by the operating system 150. Alternatively, in step 405, a hardware device, such as chip 1 220a or chip 2 220b sets interrupt line 222a or interrupt

line 222b, respectively and the operating system kernel 210 receives an interrupt.

[0046] In step 410 operating system 150 or operating system kernel 210 masks, i.e. turns off, any other interrupt lines not corresponding to the interrupt line which was set in step 405. In step 415, a series of communication between the device driver 160 and operating system 150 occurs to in order for the scheduler 152 to acquire the latency information related to the hardware device generating the interrupt. In some embodiments the scheduler 152 may compute a latency value based on the latency information. In other embodiments the latency information includes a latency value.

[0047] Alternatively, in step 415 a series of communication between the device driver 260 and the operating system kernel 210 occurs to acquire the latency information. The communication may include calls requesting allocation of an interrupt, freeing of an interrupt, scheduling of an interrupt, or the like. In step 420 the operating system 150 or the operating system kernel 210 unmasks, i.e. turns on, the interrupt lines turned off in step 410. In step 425 the scheduler 152 or a thread scheduler within the operating system dependent code 262 schedules a thread to process the interrupt received in step 405. Specifically, in one embodiment of the present invention, the interrupt handler 154 or optionally, the device driver 160, passes a latency value specified in real time units, e.g., nanoseconds, microseconds, or the like, to the scheduler 152. The scheduler 152 will, in turn, apply this thread attribute to organize all pending threads in an efficient order such that the thread and the pending threads will be processed within the latency constraints specified for each thread.

[0048] In step 430 the scheduler 152 or the thread scheduler with the operating system dependent code 262 determines if the thread scheduled in step 425 should be activated based on the latency information or an absolute time determined using a latency information, and, if not, step 430 is repeated. Otherwise, the scheduler 152 or the thread scheduler within the operating system dependent code 262 activates the thread scheduled in step 425, for example by communicating a service interrupt from the operating system 150 to

the I/O devices 120 or from the operating system dependent code 262 to the hardware device associated with the thread.

[0049] Although various embodiments which incorporate the teachings of the present invention have been shown and described in detail herein, those skilled in the art can readily devise many other varied embodiments that still incorporate these teachings. In the claims, elements of method claims are listed in a particular order, but no order for practicing of the invention is implied, even if elements of the claims are numerically or alphabetically enumerated.